FaultHound: Value-Locality-Based Soft-Fault Tolerance

Nitin, Irith Pomeranz, and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{nnitin,pomeranz,vijay}@ecn.purdue.edu

Abstract

Soft error susceptibility is a growing concern with continued CMOS scaling. Previous work explores full- and partial-redundancy schemes in hardware and software for soft-fault tolerance. However, full-redundancy schemes incur high performance and energy overheads whereas partial-redundancy schemes achieve low coverage. An initial study, called Perturbation Based Fault Screening (PBFS), explores exploiting value locality to provide hints of soft faults whenever a value falls outside its neighborhood. PBFS employs bit-mask filters to capture value neighborhoods. However, PBFS achieves low coverage; straightforwardly improving the coverage results in high false-positive rates, and performance and energy overheads. We propose FaultHound*, a value-locality-based soft-fault tolerance scheme, which employs five mechanisms to address PBFS's limitations: (1) a scheme to cluster the filters via an inverted organization of the filter tables to reinforce learning and reduce the false-positive rates; (2) a learning scheme for ignoring the delinquent bit positions that raise repeated false alarms, to reduce further the false-positive rate; (3) a light-weight predecessor replay scheme instead of a full rollback to reduce the performance and energy penalty of the remaining false positives; (4) a simple scheme to distinguish rename faults, which require rollback instead of replay for recovery, from false positives to avoid unnecessary rollback penalty; and (5) a detection scheme, which avoids rollback, for the load-store queue which is not covered by our replay. Using simulations, we show that while PBFS achieves either low coverage (30%), or high false-positive rates (8%) with high performance overheads (97%), FaultHound achieves higher coverage (75%) and lower false-positive rates (3%) with lower performance and energy overheads (10% and 25%).*

1 Introduction

CMOS scaling has resulted in increasing counts of on-chip transistors and cores in multicores. However, smaller transistors and lower supply voltages have also brought higher susceptibility to soft errors. While main memory, large on-chip caches, and on-chip networks can be protected by ECC and CRC, the cores' pipeline logic (control and datapath) are not amenable to such error-tolerant coding. Higher voltages can decrease the susceptibility but would increase energy. As such, the increasing susceptibility is one of the major impediments to voltage scaling and is a growing concern for the microprocessor industry.

While a majority of soft faults (e.g., 85%) are either masked and do not matter, or noisy (e.g., trigger address translation exceptions) and are detected easily [2], the remaining faults cause silent data corruption (SDC) which is the main challenge in soft-fault tolerance and our focus. Most of the previous proposals have employed redundant execution in hardware for soft-error detection [15, 21, 24, 27] and recovery [7, 29]. There are also software-based redundancy schemes [5, 22, 23]. However, the full-redundancy schemes incur high performance and energy overheads (our simulations show 13% and 56%, respectively). To address these overheads, other approaches advocate partial redundancy in hardware [8, 20] or software [4, 5]. However, the partial-redundancy schemes achieve low coverage [8], use aggressively-provisioned processor configurations [15], or require recompilation [4, 22].

Most previous hardware approaches perform redundant execution irrespective of the presence or absence of soft faults. A different approach is to look for hints of soft faults so that redundant execution is triggered only upon such hints, significantly reducing performance and energy overheads. In such a scheme, redundant execution via pipeline rollbacks provides recovery. The hints may be based on sudden changes in misspeculation and TLB miss rates [2] or on detecting values falling outside value locality neighborhoods [13], as explored in the initial study called *Perturbation Based Fault Screening (PBFS)* [19]. PBFS has better coverage than the former. In these schemes, however, true value changes may cause false positives which unnecessarily trigger pipeline rollbacks and result in performance and energy overheads; and true faults may go undetected resulting in loss of coverage.

Due to these issues, PBFS has a key limitation: A fundamental aspect of the value prediction-based approach is the inherent tension between coverage and false-positive rate (affecting performance and energy). PBFS employs *bit-mask filters* consisting of one-bit *sticky* counters for each bit position, to keep false-positive rates, and hence performance loss, low (1%) but at the significant cost of low coverage

ISCA '15, June 13-17, 2015, Portland, OR, USA.
 ISBN 978-1-4503-3402-0/15/06 $15.00.
DOI: http://dx.doi.org/10.1145/2749469.2750372

(e.g., 30%). Because sticky filters stay saturated until a periodic clear, they can detect only one change (true fault or false positive) per clear resulting in low coverage. Merely changing the one-bit sticky counters to well-known, two-bit, non-sticky counters improves coverage, as one might expect, but also increases the false-positive rates to the point of unacceptable performance loss (e.g., 100%). This loss remains high even if we reduce false-positive rates due to frequent value-changes by employing well-known, biased two-bit state machines [9] that signal a change only if it occurs after several no-changes.

In this paper, we propose FaultHound to make value locality-based soft fault tolerance practical by achieving much higher coverage at low overheads in performance and energy. We address the above limitations by employing five new mechanisms in FaultHound.

To address false positives and energy overhead, we make the key observation that, though nearby instructions have similar values, PBFS's PC-indexed tables needlessly spread the corresponding, similar values over multiple filters. Such spreading worsens false-positive rates because each filter has to learn individually that some values are changing based on a subset of values and the filters cannot benefit from each other. Accordingly, we propose to cluster similar values into one filter via an inverted organization of the tables using the values themselves, and not PCs, for look up. Such an organization directs similar values to the same filter rapidly reinforcing the learning of value changes. With a simple clustering algorithm, we achieve high coverage and low false-positive rates for even large, commercial workloads with just 16-32 filter entries which impose low energy overhead.

To reduce further the false-positive rates, we make the key observation that some delinquent bit-positions repeatedly trigger false positives despite our state machine's bias. Accordingly, we propose to mask such bit-positions from triggering rollbacks via a second-level filter.

For the remaining false positives, the performance and energy penalty of a full rollback is high. To reduce this penalty, we make the key observation that a full rollback can be avoided given that dependencies are usually among nearby instructions which consume values off bypass paths and not from the register file. As such, only either distant consumers, or nearby consumers upon a misspeculation/exception rollback, use the register file values; both of these cases are uncommon. Therefore, register file faults are masked usually whereas in-flight value faults get propagated to nearby consumers. Accordingly, we exploit modern out-of-order-issue pipelines' light-weight replay of instructions following a load upon the load's cache miss. We propose to replay instead of rollback upon a soft-fault trigger. However, while replay re-executes instructions following a load, a soft-fault trigger implies that a nearby preceding instruction was likely corrupted. Therefore, we propose a mechanism to replay nearby preceding instructions upon a trigger. By replacing the full rollback with our replay, we significantly reduce the performance and energy cost of false positives.

Because replay does not cover the front-end (e.g., replay would reuse the same faulty rename tag without re-executing the rename stage), we employ another mechanism in FaultHound to cover rename faults which are a significant subset of front-end faults. Because front-end stages are re-executed in current pipelines only by a full pipeline rollback, we propose to squash for rename faults. However, to avoid the performance and energy penalties of full rollbacks for every trigger, we need to distinguish between rename faults and false positives. To that end, we make the key observation that unlike a true value change, a rename fault does not directly change a value but instead causes computation to use an unintended, albeit unchanged, value. While this unintended use disrupts value locality and causes our filters to trigger similar to false positives, the unintended value also implies change in the identity of the filters that trigger unlike most false positives. We propose a simple scheme to detect this change allowing us to distinguish between rename faults and false positives.

Finally, because loads and stores wait in the load-store queue (LSQ) after execute, replay does not cover the LSQ. Therefore, FaultHound checks loads and stores at commit against the filters and re-executes them upon triggers without requiring rollbacks. However, this re-execution creates a non-intuitive problem that converts some false positives into true negatives. We address this problem via a fault detection scheme which complements the recovery achieved by our replay and rollback.

To summarize, our contributions are:

- a clustering scheme employing an inverted organization of the filter tables, where the filters use our biased state machines, to reduce the false-positive rates;
- a second-level filter for masking the delinquent bit positions, that raise repeated alarms, to reduce further the false-positive rate;
- a light-weight scheme to replay preceding instructions instead of a full rollback to reduce the performance and energy penalty of the remaining false positives;
- a simple scheme to distinguish rename faults, which require rollback instead of replay for recovery, from false positives to avoid unnecessary rollback penalty; and
- a detection scheme, which avoids rollbacks, for the LSQ which is not covered by our replay.

Using simulations, we show that PBFS achieves either low coverage (30%), or high false-positive rates (8%) with high performance overheads (97%) whereas FaultHound achieves higher coverage (75%) and lower false-positive rates (3%) with lower performance and energy overheads (10% and 25%).

The rest of the paper is organized as follows. In Section 2, we provide some background on value-locality-based soft-

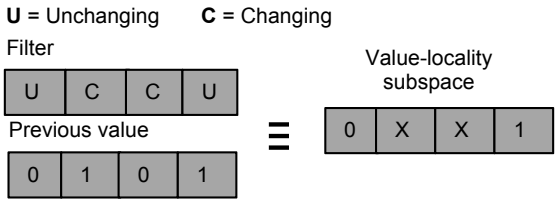

Figure 1 Value-locality space representation

fault tolerance and discuss the challenges therein. We describe FaultHound's mechanisms and operation in Section 3. We describe our experimental methodology in Section 4 and present our results in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Background and challenges

Several value prediction proposals exploit value locality for performance [3, 12, 13]. Soft error tolerance, however, is a new application for value locality which provides hints when a value strays from its locality possibly due to a fault. While value prediction must be correct for all the bits to be accurate, fault-tolerance hints need to be correct only for the faulty bit(s). Therefore, fault-tolerance hints have more leeway than value prediction.

2.1 PBFS

Because the processor, the target for soft fault tolerance, interacts with the memory system through loads and stores, checking loads and stores can cover the processor. Accordingly, PBFS checks load addresses, and store addresses and values against their respective value localities. PBFS tracks the locality of a value by using a bit-mask filter consisting of one-bit saturating counters, one per bit of the value, to indicate whether the bit changes often (the "changing" state) or not (the "unchanging" state). A change in an unchanging bit indicates a potential soft fault, with some chance of false positives, and triggers a pipeline rollback and re-execution; false positives cause unnecessary rollbacks and hence performance and energy loss. Changes in the often-changing bits are assumed to be inherent to the value and hence do not trigger any re-execution with some chance of loss of coverage. While the "changing" bit positions are wildcards which allow the bits to take on any value, the "unchanging" bit positions need to check the incoming value against the previous values of those bits. Therefore, each filter is accompanied by the previous value. Together, the filter and the previous value capture a subspace of the value space corresponding to the value's specific value locality. Figure 1 shows a 4-bit example of the filter state, the previous value, and the value subspace.

Following value prediction work, PBFS uses a table indexed by the PC where each entry holds a filter and the previous value. Upon a lookup, a match occurs when the unchanging bit positions in the filter hold the same value as the previous value. A match implies that the value is within its neighborhood and therefore does not trigger any rollback. A mismatch in one or more of the unchanging bit positions indicates a potential fault and triggers a rollback.

To keep false-positive rates low, the counters are sticky, and stay saturated at "changing" to avoid repeated false positives for values that alternate between changes and no changes. A standard, non-sticky counter would transition to the "unchanging state" when the value does not change and would incur false positives when the value changes again. The sticky counters stay saturated until a periodic, flash clear of all the filters in the table. However, by saturating to "changing" upon just one change and staying saturated, the sticky counters can detect only one change (true fault or false positive) after every periodic clear. As such, this scheme achieves low coverage (e.g., 25-30%).

Because a value change may imply a false positive or a true fault, the two cannot be distinguished. As such, the values re-computed by rollbacks are deemed final without comparing the original and re-computed values though the rollbacks may not have corrected the faults. Comparing many register values in high-speed pipelines is hard (a typical rollback squashes many tens of instructions).

While PBFS checks only loads and stores, a fault could have originated at an earlier instruction and propagated to a load or store. To increase the chances of covering such earlier instructions, PBFS squashes the pipeline immediately upon detecting a fault at load or store execution assuming that the originating instruction has not yet committed.

2.2 Challenges

As discussed in Section 1, upgrading to conventional, two-bit, non-sticky counters improves coverage, as expected, but also increases false-positive rates. This tension between coverage and false-positive rates is fundamental to such hint-based schemes. At one extreme, by triggering a rollback on every change one can achieve perfect coverage at the cost of high false-positive rates (and therefore high performance and energy overheads) and at the other, by not triggering on any change one can achieve zero false-positive rates at the cost of no coverage. The key challenge is to improve both.

The filter tables are accessed by every load and store instruction incurring high energy overhead. For example, in a 2-K entry table, each entry holds a 64-bit filter and a 64-bit previous value for a total of 32 KB, which is comparable to L1 D caches and amounts to an overhead nearly equivalent to the L1 D cache energy. Two-bit counters increase the table size to 48 KB. Shrinking the tables hurts coverage.

The high penalty of false positives due to full pipeline rollback results in unacceptable levels of performance loss. Because the fault-intolerant baseline is highly optimized for performance, unnecessary rollbacks significantly impact performance and energy. For example, assuming a baseline cycles/instruction (CPI) of 2.0, a mere 1% false-positive rate as a fraction of all instructions, with each rollback losing

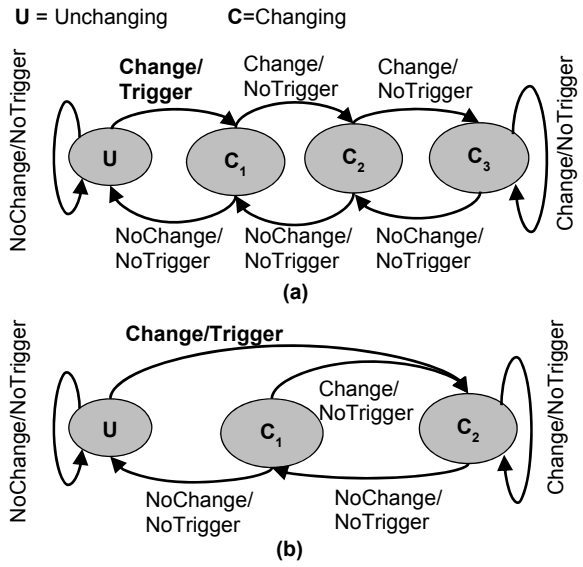

Figure 2 (a) Standard counter (b) Biased state machine

200 execution cycles (100 instructions) implies 100% slowdown (and 100% energy overhead). A key point is that the performance overhead would be lower if the baseline CPI were higher (i.e., false-positive overhead is hidden under baseline branch mispredictions and cache misses) but the energy overhead cannot be hidden.

Thus, PBFS has several limitations addressing which is challenging: (1) low coverage or high false-positive rates, (2) energy overhead of large PC-indexed filter tables, and (3) high penalty of false positives due to full rollback.

3 FaultHound

FaultHound addresses the above limitations via five new mechanisms: (1) To reduce the false-positive rates and filter energy overhead, we propose a scheme to cluster the filters via an inverted organization. (2) To reduce further the false-positive rates, we propose a learning scheme using our biased state machines to ignore the delinquent bit positions that raise repeated alarms. (3) To reduce the performance and energy overheads of the remaining false positives, we propose a light-weight predecessor replay scheme instead of a full rollback. (4) Because replay does not cover the pipeline front-end, we propose a simple scheme to distinguish rename faults, which require rollback instead of replay for recovery, from likely false positives to avoid unnecessary rollback penalty. (5) Because replay does not cover the LSQ, we propose a detection scheme, which avoids rollback, for the LSQ.

Before we describe our mechanisms, we describe the biased state machine employed by our filters. Recall that a change in the "unchanging" state triggers a rollback in PBFS. Standard saturating counters treat changes and no-changes symmetrically so that there exists direct to-and-fro transitions between a "changing" and an "unchanging" state. The direct transitions exist even if there are more "changing" states than "unchanging" states to capture the fact that true value changes are more common than faults. Figure 2(a) shows a standard counter with one "unchanging" and three "changing states" and the direct transitions between U and C_1. Bit values toggling between change followed immediately by no-change would result in repeated entry and exit out of the "unchanging" state where each exit would trigger an alarm. However, because true value changes are more common than faults, most of these alarms would be false. Therefore, we employ a well-known, biased two-bit state machine that signals a no-change more slowly than a change to reduce the false-positive rates [9]. Figure 2(b) shows the biased state machine which requires two consecutive no-changes following a change to reach the "unchanging" state whereas only one change following a no-change to reach the "changing" states. This bias slows down entry into the "unchanging" state and therefore exit from the state which triggers a rollback, reducing the false-positive rates. This bias does reduce coverage slightly because change due to a true fault in the intermediate state (C_1) does not trigger an alarm. Overall, we found that this state machine achieves good coverage and low false-positive rates (changing from two-bit to three-bit state machine reduces the coverage from 80% to 60%). Because the biased state machine is not our contribution, we include the state machine in our evaluations of PBFS. Nevertheless, as noted in Section 1, PBFS incurs high false-positive rates and performance loss.

3.1 Clustering to reduce false-positive rates

PBFS's high false-positive rates stem from the filter tables being PC-indexed. Such indexing separates nearby instructions with similar values into different table entries merely due to the instructions' differing PCs. Consequently, the filters cannot reinforce each other's state about changes in some values and instead have to learn individually about the changes in a subset of values. To address this problem, we propose to cluster similar values into one filter by using the values themselves, and not PCs, as index. Such an inverted organization directs similar values to the same filter rapidly reinforcing the learning of value changes.

A key issue with such an inverted organization is that while values are binary-encoded, the filters accompanied by the previous values essentially amount to ternary encoding: "unchanging 0", "unchanging 1", and "changing wildcard". Therefore, implementing the filter tables as a table lookup, so that an input binary value can be matched against the ternary table entries, would imply that each wildcard has to be expanded into binary values (0 and 1). However, expanding multiple wildcards would lead to an exponential explosion. To avoid this problem, we employ TCAMs (ternary CAMs) to search through all the filters instead of a table lookup. Because only 16-32 filters are needed for good coverage even for heavy-duty commercial workloads, our TCAMs remain small. As an aside, we note that TCAMs are used routinely by Internet routers for IP forwarding and packet classification [28].

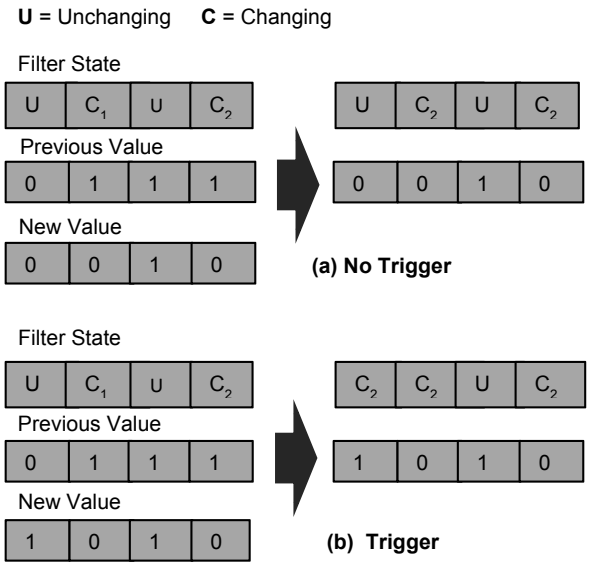

Figure 3 Filter update with (a) no trigger (b) trigger

A TCAM lookup searches for a filter that best matches the current value. As before, a match occurs for a given bit position if the state is "changing", or if the state is "unchanging" and the bit is the same in the previous value and in the current value. A match in all the bit positions of a filter means no trigger; and as part of the lookup, the matching filter's state machine is updated and its previous value set to the current value. A non-match in at least one bit position in all the filters causes a trigger (i.e., either a new value neighborhood or a fault). In this case, there is a choice for which filter to update. On one hand, installing a new filter for a non-matching value uses up a TCAM entry; and on the other, loosening (updating) an existing filter by transitioning many bit positions to "changing" for accommodating a non-matching value may make the filter miss many faults. To balance these two effects, we choose to loosen (update) the closest-matching filter as long as its non-matching bit positions are fewer than a threshold (e.g., 4) and set the filter's previous value to the current value. Otherwise (more non-matching bit positions than the threshold), we replace an existing filter with a new filter with the initial state of "unchanging" and the previous value set to the current value. As in the fully matching case, the update or replacement occurs as part of the lookup. Figure 3 shows a 4-bit example of (a) no trigger and (b) trigger, assuming a threshold of 2.

Because of the need to count the mismatches, ours is not a standard TCAM which only detects mismatches and does not provide the mismatch bit count. However, there are counting TCAMs that perform nearest-neighbor search for data mining [25].

Like PBFS, FaultHound also checks load addresses and store addresses and values. Because addresses and values often have different value locality behaviors and value ranges, mixing addresses and values weakens the filters despite clustering, worsening coverage and false-positive rates. Therefore, we employ separate TCAMs for addresses and values. While load and store addresses are instruction operands and can be identified easily, store value operands include addresses and values. As such, our TCAM for store values holds whatever store value operands contain.

3.2 Ignoring repeated false alarms

Though our clustering and biased state machine significantly reduce false-positive rates compared to PBFS our analysis showed that certain bit positions raise repeated false alarms. These delinquent bit positions undergo changes interspersed with periods of no-changes so that they toggle between the "changing" and "unchanging" states triggering false alarms upon every exit out of the "unchanging" state.

To address this problem, we employ a *single* second-level filter per TCAM (i.e., address and value) to learn the delinquent bit positions and ignore their alarms from the first-level filters. Figure 5 shows first and second-level filters (squash state machines are explained in Section 3.4). For each bit position, the second-level filter tracks whether any of the first-level filters signaled a non-match in that bit position in any of the last several times there was a replay trigger (which is a non-match due to *any* of the bits in *any* of the first-level filters). If not, a newly-occurring non-match in a bit position suggests a change in the unchanging bit position (i.e., likely a fault) and hence allows a replay; and if so, a newly-occurring non-match in a bit position is likely a false positive and is suppressed (though the state machine transitions to record the non-match).

The second-level filter employs a biased state machine with several states to slow down the signaling of an alarm for a bit (we use 8 states). This state machine requires several consecutive no-alarms before allowing an alarm for the bit (we use 7 no-alarms). While this state machine has more states than the previous, there is only one second-level filter per TCAM making the cost negligible. Overall, the second-level filter significantly reduces the false-positive rates at the cost of some modest coverage loss.

3.3 Reducing performance and energy penalty

Using full rollback like PBFS for the remaining false positives incurs high performance and energy overhead. However, a full rollback squashes all previous instructions in the pipeline so that the first instruction that was corrupted by a fault, as well as all later instructions to which the fault has propagated, are likely squashed. Thus the rollback is likely to remove the fault's effects. Fortunately, a full rollback can be avoided based on the observations that (1) loads and stores are frequent, and (2) most dependencies are among nearby instructions through which faults are propagated. Consequently, checking all loads and stores implies that a fault's propagation path is likely to be stopped not far from the first instruction that was corrupted. Therefore, squashing a short dependence chain of instructions ending in a load or store that triggers is likely to cover all of the fault's propagation path and correct the fault.

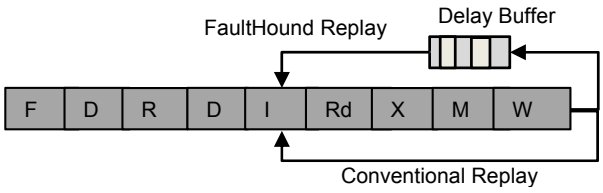

Figure 4 FaultHound replay versus conventional replay

To that end, we exploit modern out-of-order-issue pipelines' light-weight replay of instructions following a load that incurs a cache miss. To avoid complexity, conventional replay does not track whether a following instruction is dependent on the load and instead replays all following instructions between issue and memory stages of the pipeline (i.e., the pipeline back-end), as shown in Figure 4. While some independent instructions may be replayed unnecessarily, replayed instructions are few enough that the performance loss is not worth the complexity.

Upon a soft-fault trigger, we propose to replay instead of rolling back. A key issue is that while conventional replay re-executes instructions following a load, a soft-fault trigger implies the likely corruption of a nearby preceding instruction. This reversal raises three issues. First, in the absence of a normal load replay, completed instructions in conventional pipelines immediately vacate the issue queue to make room for new arrivals. As such, the instructions preceding the trigger would have vacated the issue queue by the time of the trigger in which case they cannot be replayed. To address this problem, we note that the issue queue is provisioned to overlap cache misses but cache hits are the common case. Therefore, the issue queue often has a few vacant entries [6]. Accordingly, we propose to introduce a small delay in all completed instructions' exit of the issue queue though newly-arriving instructions needing space in the issue queue can replace the completed instructions. This delayed exit increases the chance of the preceding instructions being replayed upon a trigger. At the same time, because our delayed-exit proposal does not stall newly-arriving instructions in the fault-free case, the instruction rate and hence performance is not affected. Because our proposal is a best-effort scheme, the preceding instructions may pre-maturely exit the issue queue and miss the opportunity to be replayed, resulting in loss of coverage. We implement the delay via a small delay buffer that holds a few recently-completed instructions (e.g., 6-8) for potential replay in the near future, as shown in Figure 4. The buffer operates at the normal pipeline datapath rate of instruction completion and conceptually extends the pipeline depth *after* completion which does not add bypassing or increase branch misprediction penalties.

The second issue with the reversal is the timing of the check against the filters. In a normal load replay, instructions following the load miss are identified easily as they complete and exit the pipeline datapath *after* the load. As these instructions complete, they are prevented from vacating the issue queue and are marked in the issue queue for replay. In a soft-fault replay, however, the trigger-preceding instructions complete *before* the trigger and need to be present in the delay buffer at the time of the trigger in order to be marked for replay (explained below). Because the buffer is small, this constraint implies that the filter check has to occur upon instruction completion, like PBFS (Section 2.1).

The final issue with the reversal has to do with identifying the trigger-preceding instructions for replay among the instructions in the issue queue. Upon a trigger, as the preceding instructions exit the delay buffer, they are marked in the issue queue for replay just like under a load replay. In the absence of a trigger, the preceding instructions simply vacate the issue queue just like in the absence of load replay. If a newly-arriving instruction replaces a preceding instruction in the issue queue, the preceding instruction cannot be replayed. Further, instructions later than the replaced instruction in the delay buffer should not be marked for replay as doing so would cause the later instructions to wait forever for the replaced instruction. Such a replacement is detected based on the facts that a completed instruction is being replaced and that the only completed instructions in the issue queue are the preceding instructions. To handle such replacements, our scheme simply squashes the delay buffer, potentially losing only marginal coverage.

For the same reasons as PBFS (Section 2.1), FaultHound also deems the replayed values as final. Though the filters are updated during replay to reinforce true value changes (after either faults or false positives), any triggers during replay are ignored to avoid repeated replay triggers.

We note that our delay buffer does not affect load replays which re-execute following instructions while the buffer holds preceding instructions. Like conventional replay, our replay re-executes all the preceding instructions irrespective of whether the triggering instruction is (transitively) dependent on a preceding instruction.

While full rollback squashes many instructions (e.g., 100), our replay re-executes far fewer instructions (e.g., 7). Further, full rollback repeats the work in all the pipeline stages — fetch through complete and commit — whereas replay repeats the work in the back-end of the pipeline — issue through complete. By reducing both the number of instructions re-executed and the amount of repeated work, our light-weight replay significantly reduces the false-positive penalty in performance and energy as compared to full rollback.

3.4 Handling front-end faults

Repeating the work only in the back-end implies that replay does not cover the front-end. A fault in the front-end would remain uncorrected despite replay (e.g., replay would reuse the same faulty rename tag without re-executing the rename stage). While the front-end includes fetch, decode, rename, and dispatch into the issue queue, we examine only rename

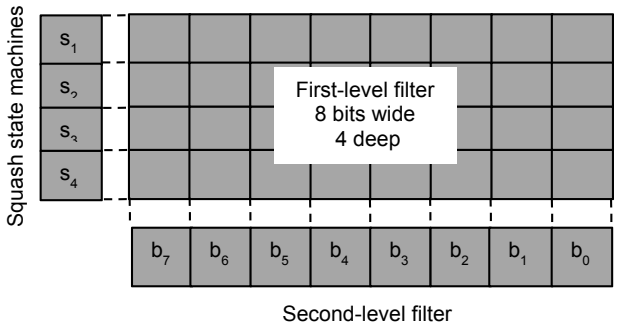

Figure 5 FaultHound's filters

faults which are a significant subset of front-end faults. Fetch largely involves the I-cache which is not addressed by FaultHound. We do not explore decode because our simulator uses a RISC instruction set (SPARC) for which decode is a small fraction of the pipeline area (under 3%).

In current pipelines, the front-end stages are re-executed only by a full pipeline rollback. Consequently, we propose to squash upon detecting rename faults. However, to avoid the performance and energy penalties of full rollbacks for every trigger, we need to distinguish between rename faults and false positives. To that end, we make the key observation that unlike a true value change, a rename fault does not directly change a value but instead causes computation to use an unintended, albeit unchanged, value. While this unintended use disrupts value locality and causes our filters to trigger similar to false positives, the unintended value also implies change in the identity of the filters that trigger (i.e., the closest-matching filters), unlike most false positives. We exploit this difference between rename faults and false positives. Inevitably, however, false-positive squash triggers would occur due to natural value locality changes which cause the identity of the closest-matching filters to change. Therefore, we need a scheme that both detects whenever the identity of the closest-matching filters change (i.e., achieve good rename fault coverage) and avoids false-positive squashes due to repeated natural value locality changes (similar to the delinquent bit positions discussed in Section 3.2).

To achieve both goals, we employ another biased state machine per first-level filter, called *squash state machine* (see Figure 5). The squash state machines are analogous to those in our second-level filter. Each squash state machine tracks whether the corresponding first-level filter was the closest-matching filter in any of the last several times there was a replay trigger from any of the first-level filters. If not, a newly-occurring trigger suggests a change in the identity of the closest-matching filter (i.e., likely a rename fault) and hence allows a squash; and if so, a newly-occurring trigger is likely a false positive and is suppressed (though the state machine transitions to record the occurrence of the trigger).

The squash state machines and those in our second-level filter are similar, except the former is one per first-level filter for all bit positions whereas the latter is one per bit position for all the first-level filters (see Figure 5). Like the second-level filter's state machine, the squash state machine allows a squash only after several consecutive no-triggers from a first-level filter (we use 7 no-triggers). Because there is only one squash state machine for all the bit positions of a first-level filter, the cost is minimal.

Thus, a trigger from a first-level filter may (1) be suppressed by the second-level filter (likely false positive), (2) cause a full pipeline rollback if signaled by the squash state machine (likely rename fault), or (3) cause a replay otherwise (likely back-end fault).

While replaying only a few instructions often correct faulty values, rolling back only a few instructions does not work for rename tags. Values are amenable to short replays by being consumed, and hence enabling the filters to trigger, usually immediately after production. In contrast, rename faults reach the filters upon instruction completion by which time many later instructions have been fetched all of which have to be squashed. Consequently, longer rollbacks are needed compared to the previous short replays.

3.5 Fault detection for load-store queue

In general, replay's redundancy does not cover the load-store queue (LSQ) because replay repeats only the work in the instruction flow before the load-store queue in the pipeline. Consequently, we separately cover the LSQ.

The approach we take is to check loads and stores at commit against the filters and re-execute upon a trigger. The filters are small and therefore fast enough to support the bandwidth demand from the checks at complete (Section 3.3) and commit, given that loads and stores are a fourth of all instructions and instruction issue rates are well under two per cycle (e.g., the LSQ is accessed at completion and commit). However, re-execution upon a trigger raises two issues. First, the check occurs at commit by which time the load or store has long exited the issue queue, ruling out our replay. A full rollback is expensive because our false-positive rates are low but non-negligible, and avoidable as we show below. Other options such as using ECC in the LSQ or a shadow copy of the LSQ values are problematic; the former is hard to implement due to tight timing and the latter increases the area and therefore fault susceptibility.

Fortunately, faults in the LSQ can be corrected by re-executing the individual load or store instruction using the values in the register file which are covered by our replay. To that end, we propose a singleton re-execute for a single load or store. The singleton re-execute suspends normal instruction select and issue for a cycle or two, and issues the load or store for re-execution. To avoid complexity, only the load or store is issued in these one or two cycles, irrespective of the normal issue width (the performance loss due to this lost issue opportunity is minimal). Because the load or store is at commit point, all previous instructions are guaranteed to have committed, and therefore completed,

implying that their values are guaranteed to have been written back to the register file and can be used by the re-execution. The original load or store is stalled at commit until the re-execution.

While the replayed values are deemed as final (Section 3.3), doing so for the singleton re-execute incurs significant loss of coverage, which brings us to the second issue which is a non-intuitive problem that the singleton re-execute may convert false positives into true negatives (i.e., loss of coverage). Any false positive in this check would trigger a re-execution well after the original completion during which time a real fault could have corrupted the load/store address or store value in the register file. Because complete-to-commit times are long (several tens of cycles), completed values in the register file have a much higher chance of being corrupted than in-flight values in the datapath. Such corruption would usually be masked due to the lack of further reads because most dependencies are among nearby instructions, as observed before, so that most consumers obtain values off bypass paths and not from the register file [17]. As such, the register file values are used only either by distant consumers or by nearby consumers upon a misspeculation/exception rollback, both of which are uncommon. Therefore, most faults in the register file are masked. Unfortunately, the singleton re-execute due to an LSQ false positive would use the corrupted value in the register file which would be deemed final and thereby convert the false positive into a true negative.

To address this subtle problem, we propose to detect such cases by comparing the re-executed values against the LSQ copy. Any mismatch implies a fault either in the register file or the LSQ and results in a fault being declared. While our replay achieves recovery in general and so does re-execution for LSQ faults in most cases (e.g., the LSQ check triggers due to a true fault), we achieve fault detection in these cases. Finally, while bulk-comparing many original and re-computed register values upon a rollback or replay is hard, the singleton re-execute lends itself to such comparisons.

4 Experimental methodology

We modify GEMS/Opal [14], simulating SPARC-based multicores running Solaris 10, to implement FaultHound and conduct fault injection experiments. We measure coverage, false-positive rates, and performance. We use McPAT [10] with GEMS for measuring energy.

Because our replay covers the register file and the back-end control and datapath, but not the front-end and LSQ (which are addressed by our other mechanisms), our fault injection must include all these structures to evaluate FaultHound's overall effectiveness. Accordingly, we inject faults into the register file, which also emulates faults in the back-end control and datapath [4, 30], and LSQ. To study the front-end, we inject rename-table faults which are a significant subset of front-end faults. As mentioned in Section 3.4, we do not study decode which is small for the SPARC instruction set.

Table 1 Benchmarks

Name	Run	Input	Warmup
SPEC2006			
400.perl 401.bzip2 429.mcf 473.astar 447.dealII 416.games 437.leslie3d	50 million instru-ctions	We use SimPoints [18] to fast-forward our simulation to a representative point and then warm up our caches for 50M instructions	
Commercial Workloads			
Apache	500 tx	20,000 files 45,000 clients 25 ms think time	20,000 tx cache 2 million system
SPECjbb	1000 tx	90 warehouses	50,000 tx cache 1 million system
Online Transaction Processing (OLTP)	40 tx	25000 warehouses 300 connections	5000 tx cache 0.1 million system
SPLASH-2 [32]			
Ocean	Full	64x64 grid	We fast forward to the inner loop and warm-up for one loop iteration
Raytrace	Full	64 MB, car.env	
Volrend	Full	inputs/head	
Water-nsquared	1 time step	216 molecules	

Using McPAT, we estimate the area of pipeline components to determine the proportions of faults injected in the rename table to represent the front-end, and in the register file and LSQ to represent the back-end (front-end 20%, back-end 80% including LSQ's 8%).

Recall from Section 1 that most faults (e.g., 85%) are either masked (i.e., do not matter), or noisy (i.e., cause exceptions) and therefore are detected. We focus on the remaining which cause silent data corruption (SDC). Because real and fault-induced exceptions are indistinguishable, we run two simulations in tandem, one fault-free and the other fault-injected. Any difference in the exceptions implies a noisy fault. To separate masked faults from unmasked ones, we inject a fault in a randomly uniformly chosen cycle during a 500-cycle period and compare the state of the two simulations after a *run-window* of 1000 instructions following the fault injection. This process assumes, similar to [19], that a fault is masked if it does not affect the processor state after the run-window. The remaining faults are binned as SDC, only for which we report coverage.

We inject 15,000 single-bit faults per run randomly over the bits in the rename table, register file, and LSQ based on the above proportions. A key issue with fault injection is that an injected fault may cause the benchmark to crash, requiring many instances of a benchmark run for the remaining injections which is slow. Instead, we use our fault-free run to update the state of the fault-injected run at the end of each run-window, catching any fault-induced exceptions as they occur and declaring the fault to be noisy. This approach allows us to use just one run for all the injections.

Our performance and energy simulations are relatively straightforward. For the performance simulations, we implemented the details of our replay in GEMS Opal. While McPAT estimates the energy dissipated in a standard processor, we use CACTI [11] to model our TCAMs' energy.

We compare FaultHound against PBFS as well as SRT, a full-redundancy detection scheme. We use the same configuration for PBFS as the PBFS paper [19] (one-bit sticky counters and 2K-entry filter tables). We choose a detection scheme, SRT, for comparison as it has lower overheads than a recovery scheme and because FaultHound is not an all-recovery scheme and has the LSQ fault detection component. We simulate an idealized version of SRT which incurs the performance and energy overheads of the trailing thread but not the leading-trailing synchronization overhead for checking loads and stores. We include the effects of key optimizations where the trailing thread incurs no branch mispredictions due to the branch outcome queue and no cache misses due to the load-value queue [21].

We draw benchmarks from four different suites – SPECint 2006, SPECFP 2006, SPLASH, and commercial workloads – to evaluate a broad spectrum of workloads. Our benchmarks are described in Table 1. We choose a mix of memory- and compute-intensive SPEC programs. Because SPEC benchmarks are sequential, we inject faults in only one core which runs two copies of the same program, one per SMT context, to simulate full load (so SRT runs four copies); for the rest of the benchmarks, which are multi-threaded, we inject faults in all the cores each of which runs two threads.

We simulate a high-performance, out-of-order-issue processor. Table 2 gives the hardware configuration parameters.

5 Experimental results

We start in Section 5.1 with a characterization of value locality and of faults. In Section 5.2, we compare FaultHound and PBFS in terms of SDC-coverage and false-positive rates. We then show the performance of FaultHound, PBFS, and SRT in Section 5.3 and their energy in Section 5.4. Section 5.5 shows a breakdown of reasons for the faults not covered by FaultHound. Finally, in Section 5.6 we isolate the impact of each of FaultHound's mechanisms.

Table 2 Hardware parameters

Processor and On-chip caches	
Cores	8, 3.2 GHz, 2-way SMT
Fetch, Decode, Issue, Commit	4 wide
ALU, Mul, FPU per core	4, 2, 2
Issue Queue size	40
Re-order Buffer	250
INT, FP arch register file	160, 64
Register windows	8
LSQ size	64
Delay buffer	7 instructions
FaultHound filters	- 2 32-entry, 64-bit TCAMs -8-state/bit second-level filter per TCAM - 8-state/TCAM-entry squash state machine per TCAM
McPAT Technology node	32 nm
Private L1 I, L1 D	32KB, 2-way, 3 cycles
Private ITLB, DTLB	64 entries
Private L2	2MB, 4-way, 20 cycles

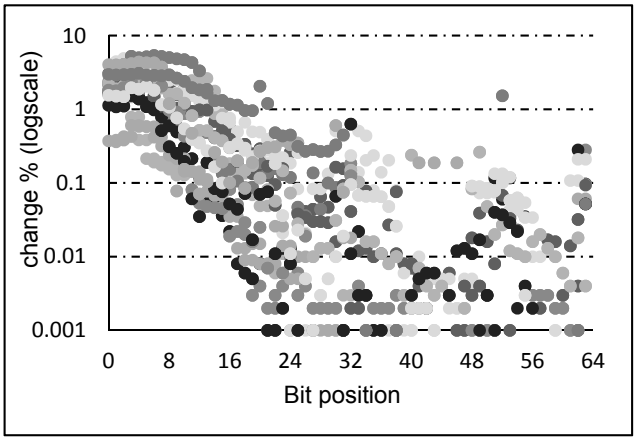

Figure 6 Percent change in bit positions

5.1 Characterization of value locality and faults

Figure 6 shows the percent of values that are different than the previous value in each bit position for load addresses, store addresses and store values. The plot shows data for all the benchmarks combined. The Y axis is in log scale. Two conclusions are: (1) Most bit positions change fewer than 1% implying high value locality so that FaultHound can achieve both high coverage (many unchanging bits) and low false-positive rates (only a few changing bits). Multiple bits of a value changing together can cause at most one trigger, implying even lower false-positive rates. On average, only 3 bits change per 64-bit write (not shown). (2) A few lower-order bit positions change more than the rest motivating the need for the second-level filter.

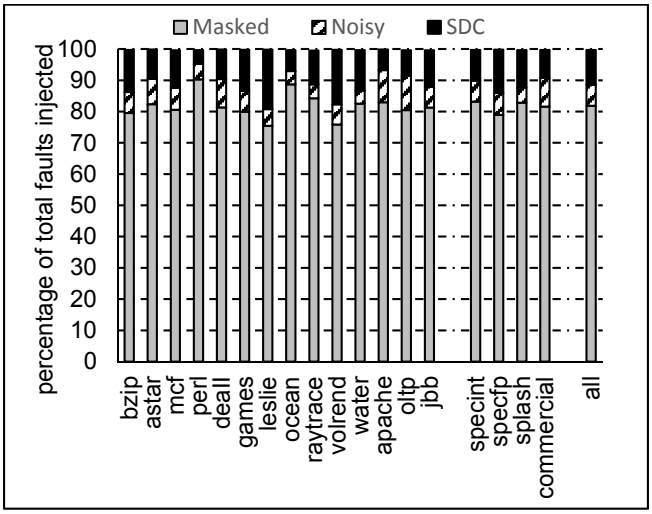

Figure 7 Fault characterization

Figure 7 shows the fraction of masked, noisy, and SDC faults out of all injected faults. The X axis shows our benchmarks, the mean for each benchmark suite and the overall mean. As discussed in Section 1, around 85% of faults are masked, around 5% are noisy (i.e., cause exceptions) and the remainder cause SDC.

5.2 Coverage and false-positive rates

Figure 8 shows the SDC coverage (a) and false-positive rates (b) of PBFS, PBFS using our biased state machine instead of the original one-bit sticky counter (PBFS-biased), FaultHound for back-end only (FaultHound-backend), and full FaultHound (Section 4 gives the proportions of faults in various pipeline components). We use the same proportions for the PBFS variants. We show FaultHound-backend in addition to FaultHound to show the impact of our replay alone. The X axis shows our benchmarks.

Recall from Section 2.1 that PBFS's sticky counters stay saturated at the "changing" state after observing one change – a false positive or a true fault. Hence, PBFS achieves nearly negligible false-positive rates but also low coverage. On the other hand, the biased state machine helps PBFS-biased achieve much better coverage but at higher false-positive rates. In contrast, FaultHound achieves similarly good coverage as PBFS-biased (Figure 8(a)) and significantly lower false-positive rates (Figure 8(b)) despite the fact that both these schemes use the biased state machines. FaultHound-backend covers only the back-end (where its coverage is reasonable) and hence achieves lower overall coverage than FaultHound. *leslie*'s low coverage across the board improves with larger filters (not shown). Compared to PBFS-biased, FaultHound improves false-positive rates by nearly 2x for SPECint and 3x for SPECFP and the commercial workloads. FaultHound's improvement in false-positive rates comes from its clustering and second-level filter, which are not present in PBFS-biased (analyzed in Section 5.6). While FaultHound's false positive rates are only slightly higher than those of FaultHound-backend, the former incurs full pipeline rollbacks due to some of the false

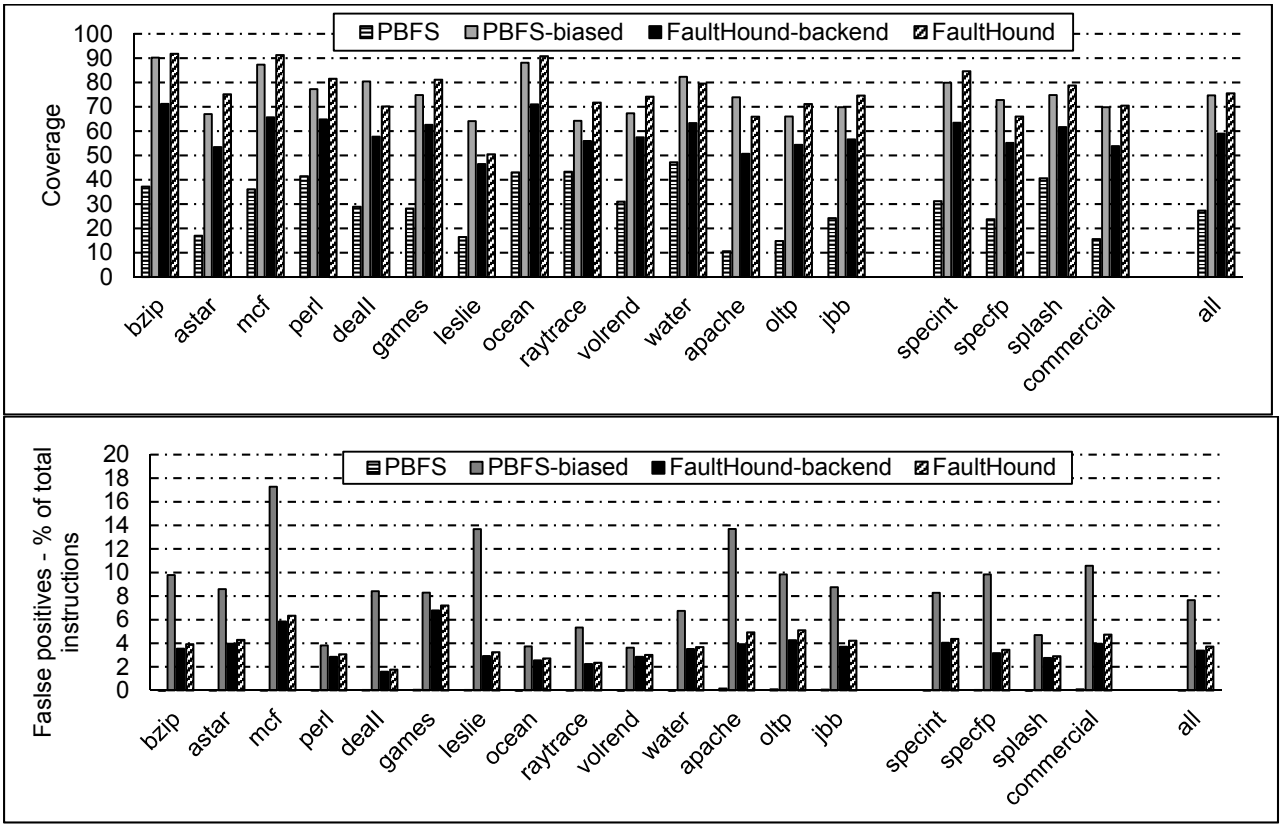

Figure 8 (a) SDC coverage and (b) false-positive rate comparison

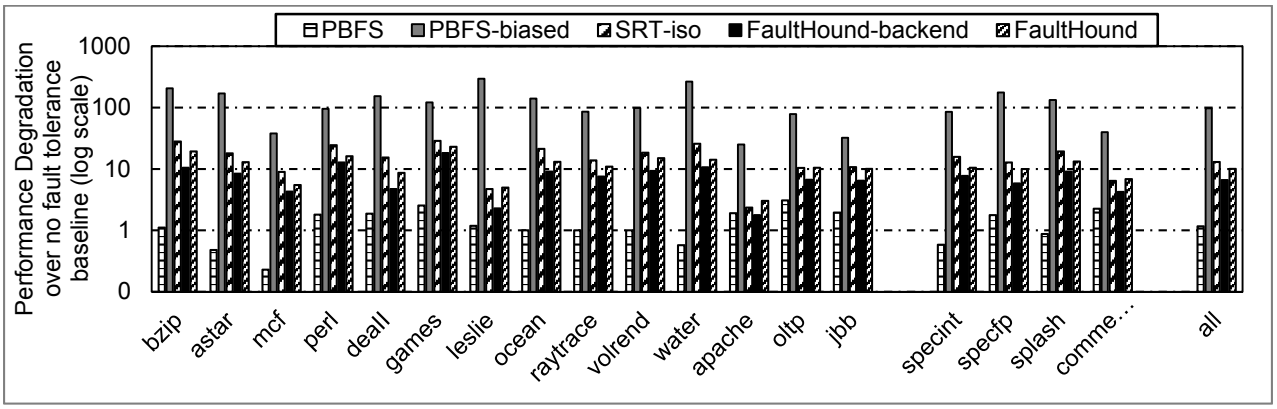

Figure 9 Performance degradation comparison

positives. These rollbacks cause significant difference in energy overheads, as we will see in Section 5.4. Overall, these results show that FaultHound achieves both good coverage and good false-positive rates, meeting the challenge stated in Section 2.2.

5.3 Performance

Figure 9 compares the performance overhead of PBFS, PBFS-biased, FaultHound-backend, FaultHound, and an SRT variant (SRT-iso) over a conventional, no-fault-tolerance baseline. The Y axis uses log scale. SRT-iso is SRT modified for partial redundancy to achieve the same coverage as FaultHound's coverage in Figure 8(a). SRT is a full-redundancy detection scheme with perfect coverage whereas FaultHound has lower coverage. To be fair to SRT, we show an SRT variant that achieves the same coverage as FaultHound by running only a fraction, equal to FaultHound's coverage in Figure 8(a), of the benchmark. To be fair to FaultHound, each core provisioned for two SMT contexts runs two threads in the baseline, PBFS, PBFS-biased and FaultHound and four threads in SRT-iso (2 original threads and their copies). Previous full-redundancy schemes, such as AR-SMT [24], SRT [21], and SRTR [29], assume that a vacant SMT context exists for the extra copy. However, unused SMT contexts imply an under-utilized system which may be unrealistic especially for server systems (SMT's purpose is to improve processor utilization). As such, we measure performance degradation over a realistic baseline.

PBFS's negligible false-positive rates imply correspondingly low performance loss; however, PBFS's coverage is low. PBFS-biased's higher coverage comes at unacceptably high performance loss due to its high false-positive rates combined with the high penalty of full rollback (around 100-200 instructions rolled back per trigger). In contrast, FaultHound-backend and FaultHound incurs under 10% performance loss in many benchmarks. Without full rollbacks, FaultHound-backend's degradation is lower than FaultHound's (at the cost of some coverage). In the commercial workloads, FaultHound's performance loss is lower because the rollback and replay overhead is dwarfed by these memory-intensive workloads' cache miss overhead. While FaultHound's low false-positive rates help keep performance loss under control, its light-weight replay helps reduce the degradation (around 6-8 instructions replayed per trigger). SRT-iso incurs slightly higher performance degradation than FaultHound due to the pressure exerted by SRT-iso's extra copies on the processor resources. This pressure exists despite SRT's optimizations of perfect branch prediction and 0% cache miss rate for the trailing thread (Section 4). In the case of the commercial workloads, SRT-iso can hide the extra copy under the cache misses of the main thread, incurring low performance degradation. In contrast, PBFS-biased's false-positive-induced rollbacks slow down the main threads' critical paths resulting in high performance loss. Therefore, SRT-iso fares much better than PBFS-biased.

Because PBFS has low coverage and PBFS-biased has high performance degradation, we analyze only FaultHound and SRT-iso in the rest of the paper.

5.4 Energy

Figure 10 shows the energy overhead of FaultHound-backend and FaultHound (including all the filters) and SRT-iso over the no-fault-tolerance baseline. While SRT-iso's extra copies' execution time can be hidden under the main copies branch mispredictions and cache misses (Figure 9), the extra copies' energy overhead cannot be hidden. Due to its significant redundancy (around 75% of committed instructions are repeated), SRT-iso incurs high energy overhead. In contrast, FaultHound-backend incurs only around 10% energy overhead due to its low false-positive rates (Figure 8(b)) as well as low replay penalty of re-executing only around 6-8 instructions per replay. FaultHound incurs full rollbacks for rename false positives which, though infrequent (Figure 8(b)), impose a significant energy overhead compared to FaultHound-backend. FaultHound's performance overhead is lower (Figure 9) for the same reasons as those for SRT-iso (as also discussed in Section 2.2).

Overall, FaultHound performs well against all three key metrics of coverage, performance, and energy, whereas PBFS, PBFS-biased and SRT-iso perform well against only one or two of these metrics.

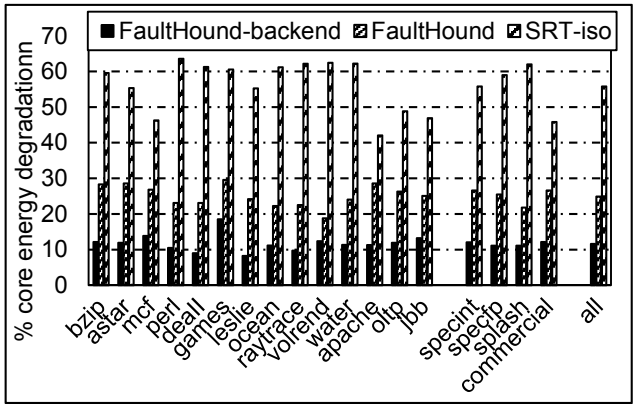

Figure 10 Energy Degradation

5.5 Fault Breakdown

To understand FaultHound's coverage, in Figure 11 we categorize the injected SDC faults as covered faults, faults masked by the second-level filter, faults in a completed or committed register, uncovered rename faults, faults that do not trigger, and other. Faults masked by the second-level filter represent the coverage cost of reducing false-positives. Replay cannot correct faults in completed and committed registers. Faults in "changing" bit positions, due either to inherent value changes or to our biased state machine, do not trigger. And, the rest of the faults are binned as "other".

In most benchmarks, the second-level filter does not induce much lost coverage. Because most values are read off bypass paths, faults in completed and committed registers account for only a modest fraction of SDC faults. To explain uncovered rename faults, we observe that unlike values which are consumed usually immediately after production, rename tags are read *much* later by over-writing instructions, well after dependent instructions execute, for freeing physical registers. While faults in values occurring after dependent instructions execute often get masked, the late reads of faulty rename tags corrupt the processor state by freeing incorrect physical registers upon commit. However, such corruption is detected after commit from which rollbacks cannot recover. The non-triggering faults constitute a prominent category considering all the benchmarks, and account for about 10% of SDC faults. To avoid false positives upon true changes, which is on average 4 out of 64 bits, the filters conservatively lose this 10% of coverage. The rest are not distinguishable by us, but are a small fraction.

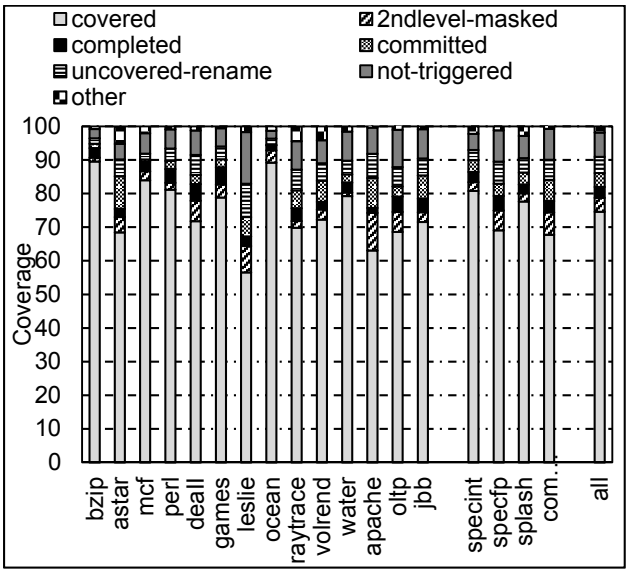

Figure 11 SDC fault breakdown

5.6 Isolation of FaultHound's mechanisms

Finally, we isolate the impact of FaultHound's mechanisms: clustering, second-level filter, predecessor replay, and covering the LSQ. Because we have already isolated the impact of our front-end scheme for rename faults from the back-end scheme, we focus only on the back-end here. Figure 12 shows three graphs and only the overall mean across all the benchmarks in the interest of space. The leftmost graph shows the impact of clustering and second-level filter on false-positive rates (Y axis), by starting with FaultHound-backend without these two mechanisms (*FH-BE-nocluster-no2level*) (i.e., similar to PBFS-biased) and adding the two mechanisms one by one (*FH-BE-no2level* and *FH-BE*). The graph shows that each of these mechanisms significantly improves false-positive rates. The middle graph shows the impact of predecessor replay on performance (Y axis) by comparing FaultHound-backend with full rollback (*FH-BE-full-rollback*) against FaultHound-backend (*FH-BE*). Because of the large difference in the number instructions re-executed (around 100-200 in full rollback versus 6-8 in replay), replay performs dramatically better than full rollback. The rightmost graph shows the impact of covering the LSQ on SDC coverage (Y axis). This graph compares coverage without this mechanism (*FH-BE-noLSQ*) and with (*FH-BE*). Covering the LSQ makes a significant difference in the SDC coverage.

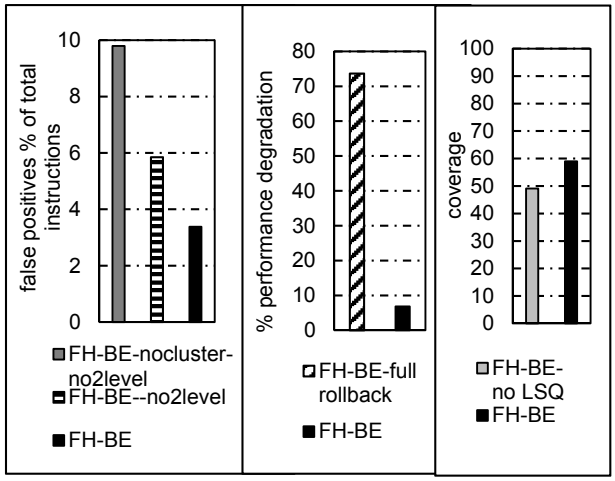

Figure 12 Isolating impact of FaultHound's back-end mechanisms

6 Related work

Soft-fault tolerance is a well-studied problem. Previous hardware proposals for soft-fault tolerance employ full redundancy via lock-stepping [1, 26] or redundant multithreading by exploiting simultaneous multithreading or multicores [7, 15, 21, 24, 27, 29]. These proposals exploit the extra resources to run two copies for fault detection or recovery. As discussed before, full redundancy incurs high performance and energy overheads. Other proposals employ partial redundancy to reduce the overheads of fault detection by triggering redundancy when there are vacant resources [6] or by introducing alterations to checkpoint and recovery mechanisms in the processor [8, 20]. However, these techniques either achieve low coverage [6] or use aggressively-provisioned processor configurations [8, 20]. Other techniques reduce soft-error vulnerability of high-performance processors [31].

Software-based redundancy schemes simplify the hardware and provide low-cost alternatives. Swift [22] carefully exploits unused instruction-level resources to introduce redundancy. Shoestring [4] exploits symptoms to avoid redundancy whenever programs exhibit symptomatic behavior of faulty computation. However, the performance and power overheads remain. Furthermore, the software schemes often require full access to application source code to enable the compiler to introduce redundancy.

Much work has been done on modeling architectural soft-error vulnerability. These include proposals for an analytical model [16, 33, 34] as well as software simulation tools [26] to quantify the architectural vulnerability of a system.

In contrast to previous work, FaultHound exploits value locality to achieve good coverage while maintaining low performance and energy overheads for typically-provisioned processors.

7 Conclusion

Increased soft-error susceptibility is an unwanted consequence of continued CMOS scaling. Previous redundancy-based schemes either incur high performance and energy overheads, or achieve low coverage. A different approach is to exploit value locality to provide hints of soft faults whenever a value falls outside its neighborhood. A previous scheme employs bit-mask filters to capture value neighborhoods. However, the scheme achieves low coverage improving which results in high false-positive rates, and performance and energy overheads.

We proposed FaultHound to address these limitations via five mechanisms: (1) a scheme to cluster the filters via an inverted organization of the filter tables to reinforce learning and reduce the false-positive rates; (2) a learning scheme for ignoring the delinquent bit positions that raise repeated false alarms, to reduce further the false-positive rate; (3) a light-weight predecessor replay scheme instead of a full rollback to reduce the performance and energy penalty of the remaining false positives; (4) a simple scheme to distinguish rename faults, which require rollback instead of replay for recovery, from false positives to avoid unnecessary rollback penalty; and (5) a detection scheme, which avoids rollback, for the load-store queue which is not covered by our replay.

Our simulations showed that the previous scheme achieves either low coverage (30%), or high false-positive rates (8%) with high performance overheads (97%), whereas FaultHound achieves higher coverage (75%) and lower false-positive rates (3%) with lower performance and energy overheads (10% and 25%). Thus, considering all the three metrics of coverage, performance and energy overheads, FaultHound performs better than previous redundancy-based and value-locality-based schemes which perform well against only one or two of these metrics. Because of this attractive combination of features, FaultHound will likely be important in the path of continued scaling.

Acknowledgments

This work was funded, in part, by the National Science Foundation under the awards numbered 1320263-CCF and 1405939-CNS. The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Data Integrity for Compaq NonStop Himalaya Servers http://www.efudan.com/course/compaq/himalaya_data_integrity.pdf.
2. ReStore: Symptom Based Soft Error Detection in Microprocessors *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, IEEE Computer Society, 2005.
3. Calder, B., Reinman, G. and Tullsen, D.M. Selective value prediction *Proceedings of the 26th annual international symposium on Computer architecture*, IEEE Computer Society, Atlanta, Georgia, USA, 1999.
4. Feng, S., Gupta, S., Ansari, A. and Mahlke, S. Shoestring: probabilistic soft error reliability on the cheap *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ACM, Pittsburgh, Pennsylvania, USA, 2010.
5. Feng, S., Gupta, S., Ansari, A., Mahlke, S.A. and August, D.I. Encore: low-cost, fine-grained transient fault recovery *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, Porto Alegre, Brazil, 2011.
6. Folegnani, D., Gonz, A. and lez. Energy-effective issue logic *Proceedings of the 28th annual international symposium on Computer architecture*, ACM, 2001.
7. Gomaa, M., Scarbrough, C., Vijaykumar, T.N. and Pomeranz, I. Transient-fault recovery for chip multiprocessors *Proceedings of the 30th annual international symposium on Computer architecture*, ACM, San Diego, California, 2003.
8. Gomaa, M.A. and Vijaykumar, T.N. Opportunistic Transient-Fault Detection *Proceedings of the 32nd annual international symposium on Computer Architecture*, IEEE Computer Society, 2005.
9. Jacobsen, E., Rotenberg, E. and Smith, J.E. Assigning confidence to conditional branch predictions *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, Paris, France, 1996.
10. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M. and Jouppi, N.P. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, New York, New York, 2009.

11. Li, S., Chen, K., Ahn, J.H., Brockman, J.B. and Jouppi, N.P. CACTI-P: architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques *Proceedings of the International Conference on Computer-Aided Design*, IEEE Press, San Jose, California, 2011.
12. Lipasti, M.H. and Shen, J.P. Exceeding the dataflow limit via value prediction *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, Paris, France, 1996.
13. Lipasti, M.H., Wilkerson, C.B. and Shen, J.P. Value locality and load value prediction *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ACM, Cambridge, Massachusetts, USA, 1996.
14. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D. and Wood, D.A. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, *33* (4). 92-99.
15. Mukherjee, S.S., Kontz, M. and Reinhardt, S.K. Detailed design and evaluation of redundant multithreading alternatives *Proceedings of the 29th annual international symposium on Computer architecture*, IEEE Computer Society, Anchorage, Alaska, 2002.
16. Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K. and Austin, T. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2003.
17. Park, I., Powell, M.D. and Vijaykumar, T.N. Reducing register ports for higher speed and lower energy *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society Press, Istanbul, Turkey, 2002.
18. Perelman, E., Hamerly, G., Biesbrouck, M.V., Sherwood, T. and Calder, B. Using SimPoint for accurate and efficient simulation *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ACM, San Diego, CA, USA, 2003.
19. Racunas, P., Constantinides, K., Manne, S. and Mukherjee, S.S. Perturbation-based Fault Screening *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, IEEE Computer Society, 2007.
20. Reddy, V.K., Rotenberg, E. and Parthasarathy, S. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ACM, San Jose, California, USA, 2006.
21. Reinhardt, S.K. and Mukherjee, S.S. Transient fault detection via simultaneous multithreading *Proceedings of the 27th annual international symposium on Computer architecture*, ACM, Vancouver, British Columbia, Canada, 2000.
22. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R. and August, D.I. SWIFT: Software Implemented Fault Tolerance *Proceedings of the international symposium on Code generation and optimization*, IEEE Computer Society, 2005.
23. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I. and Mukherjee, S.S. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, *2* (4). 366-396.
24. Rotenberg, E. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, 1999.
25. Shinde, R., Goel, A., Gupta, P. and Dutta, D. Similarity search and locality sensitive hashing using ternary content addressable memories *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM, Indianapolis, Indiana, USA, 2010.
26. Slegel, T.J., Averill, R.M., III, Check, M.A., Giamei, B.C., Krumm, B.W., Krygowski, C.A., Li, W.H., Liptay, J.S., MacDougall, J.D., McPherson, T.J., Navarro, J.A., Schwarz, E.M., Shum, K. and Webb, C.F. IBM's S/390 G5 microprocessor design. *Micro, IEEE*, *19* (2). 12-23.
27. Sundaramoorthy, K., Purser, Z. and Rotenburg, E. Slipstream processors: improving both performance and fault tolerance *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ACM, Cambridge, Massachusetts, USA, 2000.
28. Varghese, G. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., 2004.
29. Vijaykumar, T.N., Pomeranz, I. and Cheng, K. Transient-fault recovery using simultaneous multithreading *Proceedings of the 29th annual international symposium on Computer architecture*, IEEE Computer Society, Anchorage, Alaska, 2002.
30. Wang, N.J., Quek, J., Rafacz, T.M. and patel, S.J. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, IEEE Computer Society, 2004.
31. Weaver, C., Emer, J., Mukherjee, S.S. and Reinhardt, S.K., Techniques to reduce the soft error rate of a high-performance microprocessor. in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, (2004), 264-275.
32. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A. The SPLASH-2 programs: characterization and methodological considerations *Proceedings of the 22nd annual international symposium on Computer architecture*, ACM, S. Margherita Ligure, Italy, 1995.
33. Xiaodong, L., Adve, S.V., Bose, P. and Rivers, J.A., Architecture-Level Soft Error Analysis: Examining the Limits of Common Assumptions. in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, (2007), 266-275.
34. Xiaodong, L., Adve, S.V., Bose, P. and Rivers, J.A., SoftArch: an architecture-level tool for modeling and analyzing soft errors. in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, (2005), 496-505.